ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 2145–2153

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

On lazy evaluation as a tool to optimize the efficiency of large scale numerical simulations in Python

L. Gross[1], A. Amirbekyan, J. Fenwick, L. Gao, A. Mohajeri, H. Muhlhaus

Earth Systems Science Computational Center,
School of Earth Sciences,
The University of Queensland,
St. Lucia, QLD 4072, Australia.

Abstract

Scientists working on mathematical models want to concentrate on the design of models. They pay little attention to numerical methods such as the finite element method (FEM), their implementation and parallelization. The *escript* module in *python* provides an environment in which scientists can define new models using a language of partial differential equation (PDE) and spatial functions which is natural for the formulation of continuous models. This approach defines a high level of abstraction from the underlying data structures and frees modelers from issues of optimized implementation and parallelization. In its current implementation *escript* evaluates expressions which define PDE coefficients immediately for all nodes or elements of an FEM mesh. In the paper we will demonstrate that for complex rheologies such as the Drucker–Prager plasticity model, the memory requirements for this strategy are the limiting factor for scaling up the mesh size. The *python* module is backed by an *escript* C++ library where the processing is performed. We will discuss an new extension to the PDE coefficient handling provided by this C++ library which uses a lazy evaluation technique and will demonstrate the efficiency of this new extension in terms of compute time and memory usage for a practical engineering application, namely the simulation of elastic-plastic, saturated porous media.

Keywords:
Python Programming Language, Partial Differential Equations, Finite Element Methods.

1. Introduction

The *escript* package [1] is a *python* module [2] for implementing mathematical models based on partial differential equations (PDEs). It provides scientists with an easy-to-use and flexible framework to specify complex, non-linear and coupled PDEs using *python* scripts. The PDE is solved by an appropriate solver library while *escript* handles the definition and evaluation of the PDE coefficients. Abstraction from the data structures allows simulations to be

Email addresses: l.gross@uq.edu.au (L. Gross), a.amirbekyan1@uq.edu.au (A. Amirbekyan), joelfenwick@uq.edu.au (J. Fenwick), l.gao@uq.edu.au (L. Gao), arash.mohajeri@uqconnect.edu.au (A. Mohajeri), h.muhlhaus@uq.edu.au (H. Muhlhaus)
[1]Corresponding author.

1877-0509
doi:10.1016/j.procs.2010.04.240

independent from the PDE solver library and the underlying computer architecture. Modelers can run simulations on their desktop PCs as well as massive parallel supercomputers without costly modifications to the program code. A variety of applications have successfully used *escript* including earthquakes [3], Earth mantel convection [4] and volcanic lava [5].

For reasons of better flexibility and ease of use for inexperienced users, traditional PDE solving environments such as ELLPACK [6] and FASTFLO [7] are designed with their own scripting languages rather than requiring the use of low level languages like C. As a *python* module, *escript* has the benefits of a scripting language without users needing to learn another new language. In *escript*, computationally expensive calculations such as the assembly and solution of a system of linear equation are entirely implemented in C/C++ to optimize performance. However, the ease of use comes at some cost for the overall performance. In many cases, this is acceptable as the drastic reduction of development time for simulation codes significantly improves the project productivity in particular in a research environment.

In the current *escript* (version 3.0), an operation within an expression defined on the *python* level is evaluated immediately for all elements in the finite element mesh on the C/C++ level. Typically the costs for the evaluation of PDE coefficients can be neglected in comparison to the overall compute time which is clearly dominated by the solution of the linear systems. However, in this paper we will discuss an example for which the overhead from this evaluation strategy is significant and in fact dominates the memory cost for the FEM solver. So the problem size is restricted by evaluation of coefficients rather than the solver.

In order to eliminate this limitation in *escript* we use the functional programming strategy of *lazy evaluation* [8, 9] to compute PDE coefficients. Lazy evaluation means that expressions are not evaluated until their values are required. That is, expressions can be assigned to variables without actually computing the value of the expressions. In *escript*, lazy evaluation is a feature of our objects and not of the python language as a whole.

Our use of lazy evaluation means that expressions can be evaluated on a per element basis during the assemply of the global matrix without storing intermediate results for all elements at any point in time. This evaluation process operates entirely on the C/C++ level and is taylored to efficiently incorporate PDE coefficients defined by complex expressions for large finite element meshes into the assembling process. Existing simulation scripts using *escript* can easily make use of this new technique by setting a switch at the beginning of the simulation. In order to avoid the storage of large volumes of data our approach does not automatically cache intermediate results as it is typically done in the functional programming approach. However, significant performance improvements can be achieved when the user explicitly request evaluation for critical variables which are reused in the next time or iteration step (typically less then five variables even in a complex simulation).

The paper is organized as follows: In the next section we will describe a model problem, namely elastic-plastic saturated porous media flow. We will give a short overview of *escript* concepts, then show how the model problem is implemented in *escript* and will discuss the limitation of the current approach for the model problem. In Section 4 we will present our new approach in more detail and in Section 5 will show memory and timing measurement for the model problem to demonstrate its efficiency. Finally we will draw some conclusions.

2. Modelling Elastic-Plastic Saturated Porous Media

In this section, we present the model for an elastic-plastic, saturated porous media as an example of a more complex model that can be solved using *escript*. The solution algorithm we outline is suggested in the literature for practical applications, e.g. see [10]. To simplify the presentation we ignore boundary coinditions. The state of the media is described by the velocity v, stress σ_{ij} and the pore pressure q over a domain Ω.

From Darcy's law and the mass conservation law one derives the diffusion equation. By applying the backward Euler scheme for the time step size dt, the pore pressure q is updated from the previous time step q^- by the solving the PDE

$$\frac{1}{K_U - K} q - \left(dt \frac{k}{\mu_f} q_{,i} \right)_{,i} = \frac{1}{K_U - K} q^- + dt \cdot v_{j,j} \tag{1}$$

where for any function Z, $Z_{,i}$ denotes the derivative of Z with respect to x_i and Einstein's summation convention is applied. The parameter k is the soil permeability, μ_f is the pore fluid viscosity, K_U is the undrained bulk modulus and K is the bulk modulus of the porous media.

To calculate the displacement increment $du = dt \cdot v$, one has to solve the PDE:

$$-(S_{ijkl}\, du_{k,l})_{,j} = \left(\sigma^{-}_{ij} - dq\ \delta_{ij}\right)_{,j} \tag{2}$$

where $dq = q - q^{-}$ is the pore pressure increment and S_{ijkl} is the tangential tensor which depends on the rheology in use. The argument of the divergence expression on the right hand side is the stress σ^{-}_{ij} from the previous time step. For the Drucker–Prager Model, the stress is updated using the following procedure: First the new stress state σ^{e}_{ij} due to elastic deformation is calculated:

$$\sigma^{e}_{ij} = \sigma^{-}_{ij} + 2G\ D_{ij} + ((K - \frac{2}{3}G)D_{kk} - dq)\delta_{ij} + \sigma^{-}_{ik}W_{kj} - W_{ik}\sigma^{-}_{kj} \tag{3}$$

where G is the shear modulus, and D_{ij} and W_{ij} are the strain and spin are defined as the symmetric and non-symmetric part of the gradient of the displacement increment du:

$$D_{ij} = \frac{1}{2}(du_{i,j} + du_{j,i}) \text{ and } W_{ij} = \frac{1}{2}(du_{i,j} - du_{j,i})\ . \tag{4}$$

The yield function F is evaluated for the elastic stress as

$$F = \tau^{e} - \alpha \cdot p^{e} - \tau_{Y} \text{ with } p^{e} = -\frac{1}{3}\sigma^{e}_{kk} \text{ and } \tau^{e} = \sqrt{\frac{1}{2}(\sigma^{e}_{ij})'(\sigma^{e}_{ij})'} \tag{5}$$

with the deviatoric stress $(\sigma^{e}_{ij})' = \sigma^{e}_{ij} + p^{e}\delta_{ij}$. The values τ_Y and α are the yield stress and the friction parameter respectively, both of which are given as a function of the plastic shear strain $\gamma^{p} = \gamma^{p-} + \dot{\lambda}$. The plastic multiplier $\dot{\lambda}$ is set as

$$\dot{\lambda} = \chi \frac{F}{h + G + \alpha K} \text{ with } \chi = \begin{cases} 0 & \text{for } F < 0 \\ 1 & \text{else} \end{cases} \tag{6}$$

where the factor χ marks when the yield condition is violated and $h = \frac{d\tau_Y}{d\gamma^{p}}$ is the hardening modulus. The tangential tensor S_{ijkl} for the next update step for the Drucker-Prager plasticity model is given by

$$\begin{aligned} S_{ijkl} \quad = \quad & G(\delta_{ik}\delta_{jl} + \delta_{jk}\delta_{il}) + (K - \frac{2}{3}G)\delta_{ij}\delta_{kl} + \frac{1}{2}(\delta_{ik}\sigma^{-}_{lj} - \delta_{jl}\sigma^{-}_{ik} + \delta_{jk}\sigma^{-}_{il} - \delta_{il}\sigma^{-}_{kj}) \\ & + \sigma^{-}_{ij}\delta_{kl} - \sigma^{-}_{il}\delta_{jk} - \frac{\chi}{h + G + \alpha^2 K}\left(\frac{G}{\tau}\sigma'_{ij} + \alpha K\delta_{ij}\right)\left(\frac{G}{\tau}\sigma'_{kl} + \alpha K\delta_{kl}\right) \end{aligned} \tag{7}$$

3. The *escript* module

Obviously, the solution of the two linear PDEs (2) for velocity and (1) for pressure are the key components when implementing the model problem outlined in the previous section. Typically the modeler developing the model will focus on how the coefficients of PDE are set as they define the outcome of the model. In the following sections we will briefly outline the basic concept of the *escript* module [1] and will show how the model problem can be implemented in *python* using the *escript* module.

3.1. Linear Partial Differential Equations

The keystone of the *escript* environment is the `LinearPDE` class in *python* which provides the interface for the user to define and solve a general, second order, linear PDE. The general form of a PDE for an unknown vector-valued function u_i represented by the `LinearPDE` class is

$$-(A_{ijkl}u_{k,l} + B_{ijk}u_k)_{,j} + C_{ikl}u_{k,l} + D_{ik}u_k = -X_{ij,j} + Y_i\ . \tag{8}$$

The coefficients A, B, C, D, X and Y are functions of their location in the domain. Moreover, natural boundary conditions of the form

$$n_j\ (A_{ijkl}u_{k,l} + B_{ijk}u_k - X_{ij,j}) = y_i \tag{9}$$

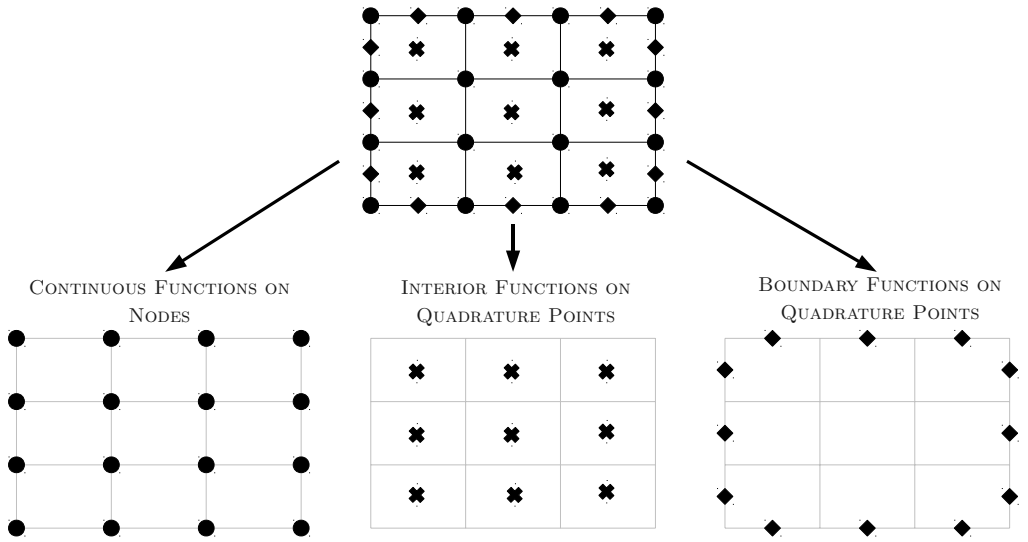

Figure 1: Location of Data-points for Selected `FunctionSpace` Types on a Finite Element Mesh.

can be defined. In this condition, (n_j) defines the outer normal field of boundary of the domain and y is a given function. To restrict values of u_i to given values, r_i, on certain locations in the domains one can define constraints of the form

$$u_i = r_i \text{ where } q_i > 0 \,, \tag{10}$$

and q_i is a characteristic function of the locations where the constraint is applied. For instance to implement the PDE for the update of the of the pore pressure in equation (1) the user needs to write the *python* script

```
from esys.escript.linearPDE import LinearPDE
mypde=LinearPDE(mydom)
mypde.setValue(A=dt*k/mu_f*kronecker(mydom), D=1./(K_U-K),
               Y=1./(K_U-K)*q+div(du))
q=mypde.getSolution()
```

where `kronecker` denotes the Kronecker δ_{ij} symbol. In this script the object `mydom` is a `Domain` class object and defines the domain of the PDE. Instances of the `Domain` class (in practice of a `Domain` subclass) are created through a function of the PDE solver library to be used to solve the PDE. For instance, the following *python* code

```
from esys.finley import ReadMesh
mydom=ReadMesh("mesh.fly")
```

creates the `Domain` class object `mydom` represented by the finite element (FEM) mesh in the file `mesh.fly` for the *finley* solver library [11]. The *finley* library reads the mesh and distributes the mesh data across the available processors if a parallel computer is used. It is beyond the scope of this paper to discuss the `Domain` class in detail. However, note that a `Domain` class object contains information about the discretization method and the PDE solver library including the distribution of FEM mesh items such as nodes and elements.

3.2. Spatial Functions

From the modeler's point of view, the coefficients of the PDE and the solution of the PDE are functions of their locations in the domain of interest. The design of *escript* adopts this view point by introducing the concept of functions of a spatial variable. These functions are represented by their values on a set of so-called data-points where the choice of the data-points depends on the type of function to be represented. The type is described by the `FunctionSpace`

class object. As shown in Figure 1 data-points can be selected as the nodes of a finite element mesh to represent continuous function on the domain, as quadrature points in the interior of the domain to represent function which may be discontinuous across element boundaries or as quadrature points on the boundary to represent functions which are defined on the boundary only. In *escript*, functions can be combined through arithmetic expressions where the results inherit the associated `FunctionSpace` from their arguments.

The stress update step described in Section 2 is written in *escript* in the following method

```
from escript import *
def getStress(du, dq, sigma, gamma_p, tau_Y, G, K, alpha, beta, h):
  g=grad(du)
  D=symmetric(g)
  W=nonsymmetric(g)
  s_e=sigma+2*G*D+((K-2./3*G)*trace(D)-dq)*kronecker(3) \
           +2*symmetric(matrix_mult(W,sigma)
  p_e=-trace(s_e)/3.
  s_e_dev=deviatoric(s_e)
  tau_e=sqrt(1./2)*length(s_e_dev)
  F=tau_e-alpha*positive(p_e)-tau_Y
  chi=whereNonNegative(F)
  l=chi*F/(h+G+alpha*beta*K)
  tau=tau_e-l*G
  sigma=tau/tau_e * s_e_dev - (p_e+l*alpha*K) *kronecker(3)
  gamma_p=gamma_p+l
  return sigma, gamma_p
```

The arguments `du` and `dq` which define the current displacement and pore pressure increment are solution of a PDE and are represented by their values at the FEM nodes. The stress `sigma` and plastic shear strain `gamma_p` which are updated by this method are represented by their values at the quadrature points in the interior of the domain. The calculation of the gradient `g=grad(du)` leads to a change in the data-points to be used for representation. In fact, the gradient is calculated by interpolating the displacement increment `du` given on the nodes and then evaluating the derivative of the interpolation polynomial on the quadrature points on an element-by-element basis. Note that in the expression `(K-2./3*G)*trace(D)-dq`, different data-points for representation are used for `D` and `dq`. In this case *escript* automatically performs an interpolation of `dq` to the quadrature points.

Typically the material parameters such as the yield stress `tau_Y` in the `getStress` method are floating point numbers. However, for composite materials the material parameters become piecewise constant functions of their location in the domain. For this case, *escript* provides a special type of representation called tagging. At mesh generation time, each individual element is marked by a tag according to its material type present at the location of the element. A piecewise constant function is represented by a dictionary mapping tags to values. In a typical simulation scenario where a very large number of elements are used to represent a composition of a small number of materials the tagged representation, if appropriate, is obviously computationally more efficient than the general expanded representation where each data-point is holding an individual value. Expanded, tagged and constant representations can be combined in a single expression where behind the scenes *escript* uses the appropriate interpretation. As pointed out earlier, this may include interpolations to another set of data-points. From the perspective of an *escript* user, this is a key feature as it allows the user to develop complex and coupled mathematical models without worrying about what type of model parameters may be fed into the model at a later stage.

4. Lazy Evaluation

To calculate the solution of the PDE, a linear system $Mu = f$ needs to be solved where u represents the solution at the nodes of the FEM mesh, f is the right hand side and M is the so-called stiffness matrix. The right hand side and the stiffness matrix are assembled from the PDE coefficients and the shape functions N used to interpolate the

solution from their values on the FEM nodes across the elements. If we focus on the coefficient A of the PDE, then the assembly of the stiffness matrix entry $M_{(i,\mu)(k,\lambda)}$ is given as

$$M_{(i,\mu)(k,\lambda)} = \int_\Omega A_{ijkl}(x) N_{\mu,j} N_{\lambda,l} \, dx = \sum_e \int_{\Omega^e} A_{ijkl}(x) N_{\mu,j} N_{\lambda,l} \, dx = \sum_e m^e_{(i,\mu)(k,\lambda)} \tag{11}$$

where i and k run through the number of solution components and μ and λ through the nodes of the FEM mesh. The value $m^e_{(i,\mu)(k,\lambda)}$ is the local element matrix for an element Ω^e. The element matrix is calculated using a numerical integration scheme

$$m^e_{(i,\mu)(k,\lambda)} = \int_{\Omega^e} A_{ijkl}(x) N_{\mu,j} N_{\lambda,l} \, dx \approx \sum_q A^e_{ijkl}(q) \, N^e_{\mu,j}(q) N^e_{\lambda,l}(q) w^e(q) \tag{12}$$

where w^e are the integration weights and index q refers to the quadrature point. The assembly process is performed on an element-by-element basis requiring a very small amount of additional memory to calculate the derivatives of the shape functions on an element. The values $A^e_{ijkl}(q)$ for all quadrature points q are accessed when element e is processed only and are typically not used elsewhere in the simulation.

The following method defines the tangential operator for the Drucker–Prager update step defined in Equation (7):

```
from escript import *
def getTangentialTensor(sigma, tau_Y, G, K, alpha, beta, h):
   k3=kronecker(3)
   pp=positive(-trace(sigma)/3)
   s_dev=deviatoric(sigma)
   tau=sqrt(1./2)*length(s_dev)
   chif=whereNonNegative(tau-alpha*pp-tau_Y)/(h+G+alpha*beta*K)*tau**2
   sXk3=outer(sigma,k3)
   k3Xk3=outer(k3,k3)
   S= G*(swap_axes(k3Xk3,0,3)+swap_axes(k3Xk3,1,3)) + (K-2./3.*G)*k3Xk3 \
    + 0.5*( swap_axes(swap_axes(sXk3,0,2),2,3) - swap_axes(sXk3,1,2)   \
          - swap_axes(swap_axes(sXk3,0,3),2,3) + swap_axes(sXk3,1,3) ) \
    + sXk3 - swap_axes(swap_axes(sXk3,1,2),2,3)   \
    - outer(chif*(G*s_dev+tau*beta*K*k3),G*s_dev+tau*alpha*K*k3)
   return S
```

The tangential tensor `S` has rank four. As it is derived from the stress `sigma` it is represented by its values at the quadrature points in each element and can be used directly in the assembly process for A (see Equation (12)). With 8 quadrature points per element and $3^4 = 81$ tensor entries per data-point, the storage of the tangential tensor for a three dimensional geometry requires at least 648 floating point numbers per element. So for a mesh with $100,000$ elements one needs about half a gigabyte just to store the PDE coefficient A alone. However, this is a very optimistic estimate for the memory requirements as it ignores the necessity to create temporary results.

To give an idea on the overhead let's have a look at a simplified version of the calculation of the tangential tensor:

```
sXk3=outer(sigma,k3)
S= sXk3 - swap_axes(swap_axes(sXk3,1,2),2,3)
```

The *python* interpreter executes these statements using two temporary variables `t1` and `t2` in the form

```
sXk3=outer(sigma,k3)
t1=swap_axes(sXk3,1,2)
t2=swap_axes(t1,2,3)
S= sXk3 - t2
```

In the best case scenario when the temporary variable `t1` is deleted after the calculation of `t2` this operation will require the storage of at least three tensors of rank four at the time where `S` is calculated The total memory cost per element is 1944 floating point numbers. For a mesh of $100,000$ elements this adds up to over 1.5 gigabytes to

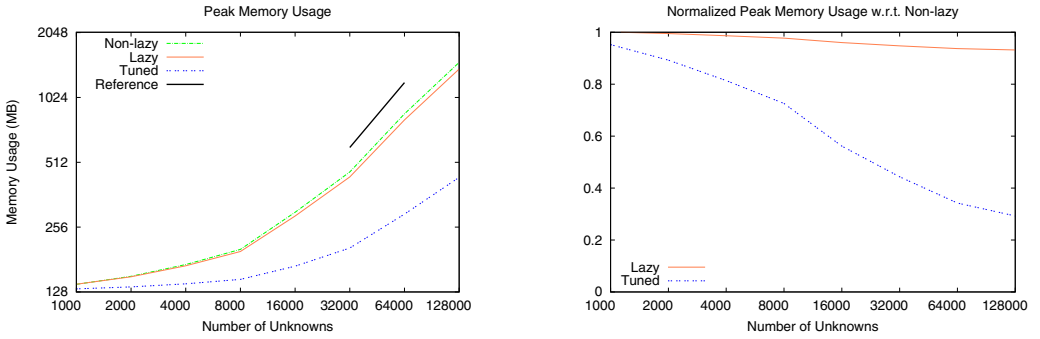

Figure 2: Peak memory used for a simulation for `lazy` and `tuned` runs relative to `non-lazy` runs. Memory includes the memory for storage of the systems of linear equations to be solved. The line marked as `Reference` shows the slope of the theorical linear growth of the peak memory with the number of unknowns.

construct the tangential operators which is about five times more than required to store the coefficient matrix of the linear system to be solved. This means that the limiting resource factor for the simulation is the assemblage process rather than solving the linear system as it is the typical situation in *escript* based simulations.

In almost all cases tensors of rank four are used once for the assembly process only. Consequently the best approach for reducing the memory requirement is not to compute the tensors for all elements at once but to build the value for a specific element at the time the value is required and then discard it. To implement this strategy, *escript* can store a function as the expression used to define it rather than its explicit value at each data-point. An expression will only be evaluated when it is actually required (hence "*lazy evaluation*") for example as part of the assembly process in Equation 12 or as a subexpression of another object. If an expression is evaluated at a given element e, each subexpression is evaluated for the particular element e and the calculated value and the identity e of the corresponding element are recorded in a small buffer. This way if a subexpression has already been evaluated for the current element e its value does not need to be recomputed. If the recorded element does not match the current element e, then the stored value is replaced. In our example, the tensor `sXk3` is used twice in the calculation of S, once directly and once in the calculation of the temporary variable `t1`. The first time a value of `sXk3` is requested for an element it will be computed, while the second time it will be returned directly from the buffer. Significantly, with lazy evaluation, the memory cost is reduced to approximately 16KB (excluding storage for inputs) independent of the number of elements.

In *escript*, the *python* interpreter provides a convienient environment in which to manipulate expressions. The data structures and their manipulation are implemented entirely in C/C++. When evalation is requested from within the solver library, it is executed within the *escript* C++ library. Values are exchanged via C pointers.

5. Timings

The *escript* implementation of the model problem as described Section 3 has been tested for three-dimensional meshes with varying numbers of elements. As the timings for the PDE coefficient evaluation does not depend on the mesh structure a rectangular mesh is used in the tests. All runs have been performed on a single core of an SGI ICE 8200 EX node with 32 GB of RAM using the Intel C/C++ compiler version 10. First as a control, simulations were run with lazy evaluation disabled. This run sets up the baseline for our experiments to compare against. We call this run `non-lazy`. In the second test, a single line was added to the *python* script to enable lazy evaluation in the C++ part of *escript*. No other changes were made to the code. This is to show the effect of using lazy evaluation with minimal effort on the part of the user. This type of test is labelled as `lazy`. When lazy evaluation is enabled in this form it is used for all expression involving *escript* functions so lazy evaluation is not just used in the calculation of the tangential tensor. As expressions are defined in terms of values from previous iteration steps, eg. `sigma` and `gamma_p` expressions become significantly larger and hence more expensive to evaluate in later iterations where in fact

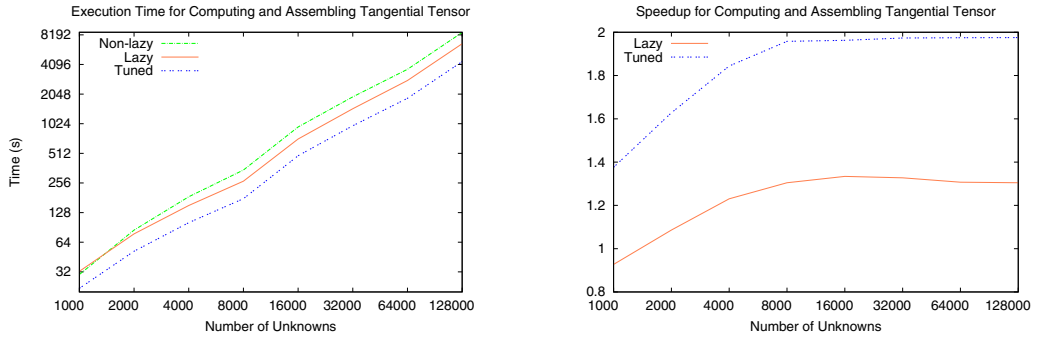

Figure 3: Execution time for computing and assembling tangential tensor and Speed-up of the execution time for `lazy` and `tuned` runs over `non-lazy` runs. Timings do not include the time for solving the linear systems.

a majority of the values need to be recalculated every time an expression is evaluated. To reduce this overhead *escript* evaluates expressions with depth greater than 70 immediately. This is the price to be paid to deploy lazy evaluation globally.

Efficiency can be improved with help from the user. As the two variables `sigma` and `gamma_p` are reused in the next iteration step an explicit evaluation of this variables is enforced after their expressions have been defined. In fact, the user needs to just add the line

```
resolveGroup((sigma,gamma_p))
```

which triggers a simultaneously evaluation of the two expressions for `sigma` and `gamma_p`. The evaluation of one of the expressions at a time would require the reevaluation of shared subexpressions, eg. of `l`. This type of evaluation is labelled as `tuned`.

Figure 2 shows the total memory used for the whole simulation for the `lazy` and `tuned` type relative to the `non-lazy` approach. As it is impractical to measure memory usage of the assembly process, only we present the total memory including memory for the systems of linear equations. The total saving in memory for the `lazy` type is in the best case about 10% for the reasons pointed out above. With the additional small code modifications a reduction of memory usage of over 70% is achieved.

Figure 3 shows the execution time of all three types of runs and the speedup of `lazy` and `tuned` runs over `non-lazy` runs for computing the tangential tensor and assembling of the stiffness matrix. The speed-up increases with the number of elements and reaches almost two for the largest instance and the `tuned` type. This significant performance improvement is caused by a dramatic reduction of cache misses while for the `lazy` type the speed-up is not as good as for the `tuned` type due to reevaluation. In the first iteration step about 35% of the execution time is spent in the expression evaluation and the assembly of the linear systems. This portion drops to 23% if the `tuned` implementation is used which leads to a reduction of the compute time by 18%. This speed-up is pessimistic as the simulation did not use the optimal linear solver.

6. Summary

We presented an extension to the *escript* C++ library which uses lazy evaluation to speed up three-dimensional *escript* simulations using *python* as a high-level programming environment. Limiting the complexity of the expressions to be evaluated is crucial to achieve optimal speed-up and to minimize memory usage. With rather moderate modifications to the *python* code (in the example discussed in this paper two line need to be added) a speed-up of a factor two and a memory reduction of 70% can be achieved. To implement the required modifications the user needs to identify variables which will be reused in the next iteration step. We believe that considering the fact that in

practice these variable will be critical state variables of the model even an unexperienced user will be able to insert the necessary modifications into the code. This implementation of lazy evaluation which will be available in *escript* version 3.1 can be used with both MPI and OpenMP parallelism.

Acknowledgements

This work is supported by the AuScope National Collaborative Research Infrastructure Strategy, the Queensland State Government and The University of Queensland.

References

[1] L. Gross, L. Bourgouin, A. J. Hale, H.-B. Muhlhaus, Interface modeling in incompressible media using level sets in escript, Physics of the Earth and Planetary Interiors 163 (2007) 23–34. doi:doi:10.1016/j.pepi.2007.04.004.
[2] M. Lutz, Programming Python, 2nd Edition, O'Reilly, 2001.
[3] L. M. Olsen-Kettle, D. Weatherley, E. Saez, L. Gross, H.-B. Mühlhaus, H. L. Xing, Analysis of slip-weakening frictional laws with static restrengthening and their implications on the scaling, asymmetry, and mode of dynamic rupture on homogeneous and bimaterial interfaces, J. Geophys. Res. 113 (2008) B08307. doi:doi:10.1029/2007JB005454.
[4] M. Davies, H.-B. Mühlhaus, L. Gross, Thermal effects in the evolution of initially layered mantle material, Journal Pure and Applied Geophysics 163 (11–12). doi:10.1007/s00024-006-0143-x.
[5] L. Bourgouin, H.-B. Mühlhaus, A. J. Hale, A. Arsac, Studying the influence of a solid shell on lava dome growth and evolution using the level set method, Geophysical Journal International 170 (3) (2007) 1431–1438.
[6] J. R. Rice, R. F. Boisvert, Solving Elliptic Problems Using ELLPACK., Vol. 2, Springer Series in Computational Software, 1985.
[7] A. N. Stokes, N. G. Barton, L. X.-L., Z. Z., Fluids, finite elements and multi-physics, NOTES ON NUMERICAL FLUID MECHANICS 85 (2003) 262–276.
[8] D. P. Friedman, M. Wand, Essentials of Programming Languages, 3rd Edition, MIT Press, 2008.
[9] A. K. Pandey, Programming Languages, Principles and Paradigms, Alpha Science, 2008.
[10] N. Manoharan, S. P. Dasgupta, Consolidation analysis of elasto-plastic soil, Computers & Structures 54 (6) (1995) 1005–1021.
[11] M. Davies, L. Gross, H.-B. Mühlhaus, Scripting high-performance earth systems simulations on the sgi altix 3700., in: Proceedings of the 7th international conference on high-performance computing and grid in the Asia Pacific region, SIGHPC / Information Processing Society of Japan (IPSJ) and IEEE Computer Society Japan Chapter, IEEE Computer Society, 2004, pp. 244–251. doi:10.1109/HPCASIA.2004.1324041.